Uma Proposta de Framework para Desenvolvimento de Aplicações Paralelas com Mobilidade

Dárlinton B. F. Carvalho[1], Carlos José P. de Lucena[1], Celso Carneiro Ribeiro[2]

Resumo: Neste trabalho propõe-se um framework orientado a objetos para o desenvolvimento de aplicações distribuídas em grades com suporte à mobilidade. O desenvolvimento é baseado no paradigma de sistemas multi-agentes, utilizando-se a plataforma JADE. A comunicação entre as aplicações é baseada no padrão MPI. A proposta deste trabalho é utilizar plataformas para desenvolvimento de sistemas orientados a agentes sem a necessidade de programá-los diretamente. Com isso, é possível facilitar o desenvolvimento de aplicações distribuídas em grades com suporte à migração dos processos entre as máquinas.

Palavras-chaves: Programação paralela, Engenharia de sistemas orientados a agentes, Mobilidade forte de processos, MPI, JADE, Java

Abstract: In this work, we propose an object oriented framework for developing distributed applications on grids supporting mobility. The system model follows the multi-agent agent system paradigm and is implemented in JADE. Communication between applications makes use of the MPI standard. We propose the use of frameworks for the development of agent-oriented systems without the need of programming them directly. Therefore, it is possible to make easier the development of distributed applications on grids adding support for process migration over the machines.

Keywords: Parallel programming, Agent-oriented software engineering, Strong process mobility, MPI, JADE, Java

1 Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro
Rua Marquês de São Vicente 255, Rio de Janeiro, Brasil
E-mail: {dbfc, lucena}@inf.puc-rio.br

2 Instituto de Computação
Universidade Federal Fluminense
Rua Passo da Pátria 156 - Bloco E - 3 andar, Niterói, Brasil
E-mail: celso@ic.uff.br

1 Introdução

O desenvolvimento de *software* orientado a agentes, com uma análise de abstrações mais expressivas e síntese de sistemas por *frameworks*, facilita o desenvolvimento de *software* para computação em grades. Diversos ambientes de sistemas multi-agentes facilitam a programação de *software* de relativa complexidade como, por exemplo, envolvendo a migração de agentes entre máquinas. Porém, nesse paradigma a programação de sistemas distribuídos mais simples se torna mais difícil pelo excesso de componentes. A proposta deste trabalho é utilizar ambientes orientado a agentes sem a necessidade de programá-los diretamente. Com isso, facilitar o desenvolvimento de aplicações distribuídas em grades com suporte à migração dos processos entre as máquinas.

O termo grade, do inglês *grid*, foi concebido na década de 90 para denotar uma proposta de infra-estrutura para computação distribuída em um âmbito mais avançado [2]. Atualmente, a computação em grade é vista como um paradigma emergente de infra-estrutura da tecnologia da informação, com características peculiares na forma de computação, comunicação e colaboração. Em outras palavras, computação em grade é uma coleção de recursos heterogêneos e distribuídos, possibilitando que sejam utilizados em grupo para executar aplicações de larga escala. Há uma expectativa de que muitas delas irão existir, cada uma com seu próprio contexto, compartilhadas por parceiros com os mesmos interesses. Alguns exemplos dessa realidade são os projetos SETI@home [2], Folding@Home [3] e o ambiente Xgrid [4] que já vem embutido no sistema operacional MAC OS™.

Uma especificação, que se tornou padrão por consenso, para o desenvolvimento de software paralelo e distribuído é o *Message Passing Interface* (MPI) [5]. MPI é uma proposta de especificação para bibliotecas que suportam troca de mensagens para ambientes paralelos, especialmente aqueles com memória distribuída. Nesse modelo, uma execução compreende um ou mais processos que se comunicam chamando rotinas de uma biblioteca para receber e enviar mensagens para outros processos. MPI foi definido por um amplo comitê composto pela indústria, implementadores e usuários, sendo projetado para ambientes de alto desempenho em máquinas paralelas e em *clusters* de estações de trabalho.

O paradigma de desenvolvimento de software orientado a agentes facilita a especificação dos sistemas por considerar abstrações mais sofisticadas, como por exemplo, os agentes, os papéis de agente, as organizações e o meio ambiente [8]. Uma outra vantagem é a utilização de plataformas para implementação de sistemas multi-agentes, que fornecem um ambiente todo preparado para suportar os comportamentos de um agente. Uma dessas plataformas é JADE [9] que, como descrito em seu *whitepaper*, é um facilitador de tecnologia, ou seja, um *middleware* para desenvolvimento e execução de aplicações ponto a ponto baseadas no paradigma de agentes e que podem trabalhar e interoperar em ambientes com fio e sem fio.

A partir desses conceitos, é proposto um *framework* que tem como objetivo facilitar o desenvolvimento de aplicações em grade agregando facilidades para uma migração automatizada de processos. Para alcançar esta meta, é adotada uma arquitetura baseada nos pilares de computação em grade composta por três componentes: aplicação, migração e gerência de recursos. O sistema de troca de mensagens baseado no padrão MPI permite uma curva de aprendizagem reduzida para o desenvolvimento de aplicações nesse novo ambiente. A migração automatizada é um componente desenvolvido à parte do código da aplicação, permitindo uma melhor separação dos conceitos relacionados. O último componente do *framework* é a gerência de recursos, que é realizada de modo específico e adaptável às peculiaridades de cada aplicação.

Este artigo apresenta primeiramente outros trabalhos relacionados com a área de desenvolvimento de sistemas multi-agentes com suporte a mobilidade e com computação em grade. Na seqüência, é dada uma visão geral do *framework*, contextualizando os conceitos envolvidos no desenvolvimento deste trabalho. Em seguida, são apresentados detalhes do desenvolvimento da aplicação orientado a agentes e um estudo de caso baseado no crivo de Erastóstenes utilizado para avaliar o *framework*. Por fim, são feitas considerações sobre este trabalho.

2 Trabalhos relacionados

Mobilidade e interoperabilidade são aspectos importantes da computação distribuída. A máquina de execução Java fornece um ambiente adequado para a execução de aplicações em diferentes máquinas. O princípio básico para dar mobilidade a processos Java é a serialização da execução de uma *thread*. Diversos projetos [12, 13, 14, 15, 16, 17, 18, 19] tratam questões sobre a serialização de *thread* em Java, embora muitos deles fujam da especificação padrão da linguagem. A maioria não apresenta evolução do software e muitos projetos já foram finalizados.

ProActive [21] é um Java *middleware* para programação paralela, distribuída e concorrente que fornece serviços de alto nível como migração fraca, comunicação em grupo, segurança, *deployment* e componentes. A proposta desse sistema está na utilização do padrão de projeto chamado objeto ativo (*active object*). Um objeto ativo pode ser definido como um objeto que possui linha de execução própria e se comunica com outros objetos através de troca de mensagem. Seu propósito de ser uma plataforma para computação de alto desempenho [22] demanda meios alternativos de comunicação entre processos, como a utilização do Ibis [24] ao invés do padrão RMI da linguagem Java. Uma comparação dessa proposta é realiza frente ao FORTRAN MPI [22], onde se obteve resultados satisfatórios. Não foram avaliadas outras implementações de MPI para Java, pois só foram avaliadas implementações com suporte total a portabilidade.

A lista de agentes móveis [20] possui cadastrados mais de 60 sistemas que suportam agentes móveis, sendo a grande maioria baseada em Java. A plataforma JADE [9] fornece mobilidade de agentes em sua especificação e foi utilizada na implementação deste trabalho.

3 O *framework*

Frameworks são geradores de aplicações que estão diretamente relacionadas a um domínio específico [1]. O domínio do *framework* proposto é o desenvolvimento de aplicações distribuídas em grade com troca de mensagens baseada em MPI, agregando facilidades para migração dos processos e utilização dos recursos.

MPI (*Message Passing Interface*) é um "padrão por consenso" de especificação de ambientes para o desenvolvimento de aplicações distribuídas, introduzido pelo MPI Forum em 1994 e atualizado em 1995. Seu projeto incluiu a participação de vendedores de *hardware*, pesquisadores, acadêmicos, desenvolvedores de bibliotecas de *software* e usuários. Este consórcio representa quase 40 organizações. A documentação oficial está disponível no site do MPI Forum [5]. Neste trabalho, considera-se apenas um conjunto das funcionalidades definidas por esse padrão. Essas funcionalidades implementadas são suficientes para programar a maioria das aplicações paralelas completas. Ainda há adaptações em relação aos parâmetros dos métodos, visto que o padrão MPI é baseado em FORTRAN, uma linguagem de programação do paradigma procedimental, enquanto o *framework* desenvolvido está fundamentado no paradigma orientado a objetos e implementado em Java. Portanto, houve a necessidade de adaptar as assinaturas das funções para compatibilizar sua utilização neste outro paradigma.

O paradigma de computação em grade é uma realidade em diversas instituições e está cada vez mais disponível para os programadores. Entretanto, há uma carência de ferramentas para facilitar a implementação de sistemas nesse contexto mais complexo. Enquanto no desenvolvimento de sistemas tradicionais de computação a preocupação está basicamente em fornecer informação, no processamento distribuído outros elementos entram em cena como gerenciamento de recursos e dados distribuídos. A Figura 1 esboça os elementos principais da computação em grade, fundamentados em segurança. A utilização de ambientes bem projetados auxilia na segurança dos sistemas a serem desenvolvidos, pois possibilita a reutilização dos elementos de segurança dessas arquiteturas. Para facilitar o gerenciamento dos recursos, no *framework* proposto o código relativo ao gerenciamento de migração fica separado do restante da aplicação. Desse modo, facilita a implementação da aplicação através de uma arquitetura preparada para este comportamento.

Plataformas para implementação de sistemas multi-agentes fornecem um ambiente preparado para suportar os comportamentos de um agente. Uma característica importante para o desenvolvimento deste trabalho é a questão da mobilidade. Considerando o tipo de migração, os sistemas com agentes móveis são classificados em duas categorias, conforme suportem mobilidade forte ou não [6]. Sistemas que suportam mobilidade forte permitem capturar e transferir o estado de execução completo do agente em migração. Sistemas que suportam mobilidade fraca apenas reiniciam a execução de um agente no sistema remoto. A maioria dos sistemas multi-agentes existentes são baseados em Java [20], citando-se alguns como JADE, Aglets, D'Agents, Voyager e Nomads. Pelo fato da especificação Sun para a máquina virtual Java (JVM) não permitir a captura do estado de execução, poucos sistemas multi-agentes suportam mobilidade forte [7]. Os que possuem esse tipo de mobilidade a

fazem através de JVM modificadas ou através de um pré-processamento do código. No entanto, o marco na evolução desses ambientes foi o lançamento da JVM 1.02, onde o mapeamento das linhas de execução (*threads*) passou a ser direto para o sistema operacional. Atualmente, os sistemas multi-agentes que continuaram em desenvolvimento adotaram abordagens arquiteturais fornecendo um modelo de mobilidade forte. Em nossa proposta, a migração forte é garantida através da captura controlada do contexto de execução da aplicação e inicialização do processamento com esse contexto.

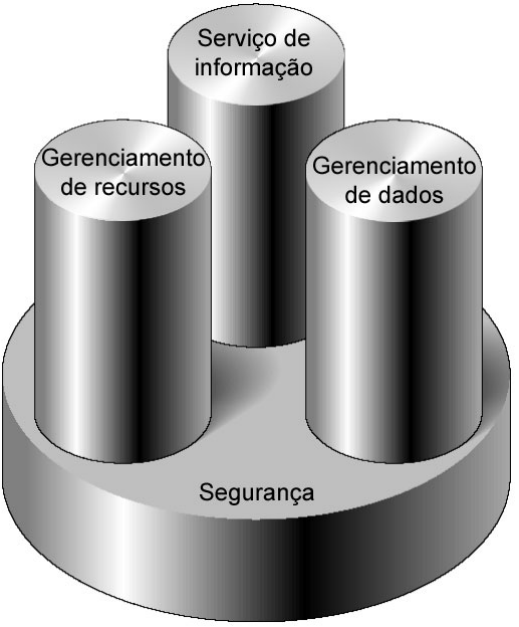

Figura 1 Os três pilares para computação em grade sobre infra-estrutura de segurança [10]

Uma vez que *frameworks* são criados para gerar aplicações completas, deve haver pontos flexíveis, que são customizados de acordo com aplicações específicas para solucionar um problema particular. Os pontos flexíveis propostos são agrupados em três componentes: a aplicação, as regras para migração e o escalonador de recursos. A aplicação consiste no programa que deverá ser executado e um procedimento para salvar os dados da aplicação. As regras para migração estão relacionadas com as condições que devem ser satisfeitas para que um processo seja migrado. O escalonador determina as máquinas utilizadas pelas aplicações.

Algumas características do *framework* estão presentes em todas as aplicações do domínio. Esses pontos imutáveis constituem o núcleo de um *framework* e são chamados pontos fixos. O núcleo é invariável e também é uma parte sempre presente em cada instância do domínio. De modo geral, os pontos fixos do *framework* proposto são o ambiente de

execução e a arquitetura do sistema. O ambiente de execução é a plataforma JADE, que executa sobre uma JVM e disponibiliza um ambiente para execução de sistemas multi-agentes. A arquitetura do sistema é definida a partir de três componentes: um ponto único responsável pelo gerenciamento dos recursos, o procedimento de migração dos processos e as funções de comunicação e de informação sobre contexto baseadas em MPI. As funções MPI implementadas são um conjunto reduzido do que é estabelecido pelo padrão, mas suficiente para programar aplicações paralelas completas. Também cabe ressaltar que as demais funções podem ser facilmente mapeadas para o suporte oferecido pela plataforma JADE. Na Figura 2, pode-se visualizar as fronteiras do *framework*.

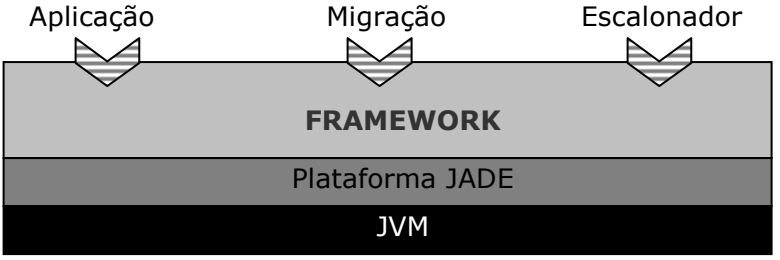

Figura 2 Visão geral do *framework*

4 Desenvolvimento orientado a agentes

Na modelagem do sistema são utilizados diagramas UML com algumas adaptações para destacar os agentes. Vemos o agente como um ator dentro do próprio sistema com autonomia de ação. São considerados dois agentes: o agente escalonador e o agente processador. O agente escalonador tem como responsabilidades iniciar a execução de uma aplicação, definir as máquinas para alocação dos processos, determinar máquinas para migração dos processos, atualizar a utilização das máquinas e encerrar a execução do ambiente. O agente processador é responsável por controlar a execução de um processo, a troca de mensagens entre os processos, verificar as condições para migração e, quando necessário e possível, realizar a migração da execução.

O diagrama de casos de uso do *framework* é apresentado na Figura 3. O ator Usuário representa a pessoa que irá disparar a execução do sistema. O ator Programa MPI representa a aplicação que será executada na grade pelos agentes processadores e utiliza os serviços providos pelo *framework*. Os casos de uso especificam os serviços disponíveis que são baseados no padrão MPI. Os casos de uso "Trocar máquina" e "Definir máquina para alocação" são relativos ao gerenciamento de recursos da grade computacional, e apenas os agentes podem utilizar essa parte do sistema. Na instanciação do *framework*, as

funcionalidades específicas para a aplicação desenvolvida, que são programadas através dos pontos flexíveis, utilizam-se dos serviços definidos pelos casos de uso.

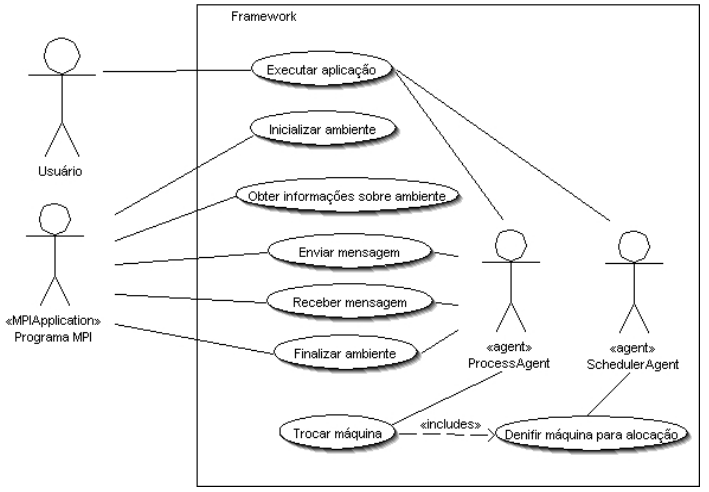

Figura 3 Diagrama de casos de uso

A implementação do *framework* é fortemente baseada em padrões de projeto [11]. Como é um *framework* orientado a objetos, para a instanciação de um sistema o usuário deve estender classes abstratas e implementar alguns métodos. Esse procedimento é baseado no padrão de projeto Template Method. A instanciação dos pontos flexíveis é feita através de um objeto do tipo da classe HotSpotFactory que é baseada no padrão Factory Method e no Singleton. Essa classe, que também é um ponto flexível, necessita de uma implementação concreta e sua criação é definida através da instanciação de uma classe com nome, obrigatoriamente, de app.ConcreteFactory. O diagrama de classes é apresentado na Figura 4.

O ponto flexível referente à aplicação é definido pelas classes MPIApplication e Memento. A aplicação MPI é vista como parte integrante do agente processador (ProcessorAgent) que é um agente da plataforma JADE, e por isso sua definição é uma classe derivada de jade.core.Agent. Quando o agente processador é iniciado (setup()), ele cria uma *thread* (SerializableThread) independente para a execução da aplicação.

A aplicação é implementada como uma classe derivada de MPIApplication, que provê acesso às funcionalidade MPI definidas pela interface MPIInterface. Informações sobre a quantidade de processadores no sistema (MPI_Comm_size()) e o número individual de cada processador (MPI_Comm_rank()), chamadas informações de contexto da aplicação, são dados locais de cada agente processador e estão prontamente disponíveis para a aplicação. Para as demais funcionalidades, foi utilizado o padrão de projeto *Adapter* para aproveitar os recursos fornecidos pela plataforma JADE. O envio e o recebimento de mensagens são

baseados na troca de mensagens entre agentes na plataforma JADE, e estão disponíveis através dos métodos MPI_Recv e MPI_Send. Ao ser chamada a função de finalizar o ambiente MPI (MPI_Finalize()), o agente processador informa ao agente escalonador que sua execução está terminada. Desse modo, o agente escalonador não precisa ficar perguntando aos agentes processadores se eles ainda estão processando a aplicação e se a máquina que estava em utilização está disponível.

Objetos da classe Memento são repositórios de dados da aplicação, como definido no padrão de projeto homônimo. Eles devem fornecer dados sobre o estado computacional da aplicação. Isso é útil para a implementação da migração da aplicação, apresentada a seguir. Portanto, para implementar o ponto flexível da aplicação deve-se definir uma classe derivada de MPIApplication, por exemplo app.MyApp, e outra derivada de Memento, por exemplo app.MyMemento.

A migração da aplicação simula um mecanismo de migração forte e está diretamente relacionada com a definição da interface MigrationAnalyzer e o comportamento MigrateBehaviour do agente processador (ProcessorAgent). O agente processador inicia a execução da aplicação e de tempos em tempos, definido pelo método getAnalyserTime(), executa o comportamento MigrateBehaviour que verifica se há necessidade de migrar a execução da aplicação. Essa verificação é realizada avaliando-se o retorno do método hasToMigrate() de MigrationAnalyzer. A aplicação, que é uma instância de uma classe derivada de MPIApplication, inicia sua execução a partir do método main() e deve periodicamente verificar se foi solicitada a interrupção de sua execução. Isso é programado através da chamada ao método isInterrupted(), e deve ser implementado na aplicação do usuário. Quando solicitada para interromper sua execução, a aplicação deve ficar em um estado adequado para migração, ou seja, em um ponto seguro onde seja possível salvar seu estado interno. O agente processador, que estava aguardando pelo término da aplicação (canMoveAgent()), pode então salvar os dados da aplicação através da criação de um memento. Isso é realizado através da chamada ao método saveState() que retorna um objeto, de uma classe derivada de Memento, contendo as informações do estado computacional instantâneo da aplicação. A migração da execução do agente processador de máquina para outra é realizada através de uma funcionalidade disponibilizada pela plataforma JADE. Após mudar de máquina (afterMove()), o agente processador restabelece a execução da aplicação configurando seu estado computacional com o memento armazenado (restoreState()) e invocando o método restart() da aplicação do usuário. Pode haver tantas migrações quantas forem necessárias.

O ponto flexível de migração é implementado através de uma classe que estende MigrationAnalyser, por exemplo app.MyMigrationAnalyser. A migração do processamento da aplicação é realizada sempre que o método hasToMigrate() retornar verdadeiro. Essa verificação é realizada periodicamente com intervalo de tempo determinado pela função getAnalyserTime(). A lógica para determinar se uma migração deve ser realizada é particular de cada aplicação, por isso ficando em aberto para seu desenvolvedor.

O ponto flexível do escalonador está proposto de um modo simplificado através de funções para obter e liberar *containeres* de execução (IScheduler) relacionadas com o agente escalonador (SchedulerAgent). Cada agente processador é executado em um *container* na plataforma JADE. Cada *container* está localizado em uma máquina diferente, por isso, um agente mudar de máquina é o mesmo que mudar de *container*. O desenvolvedor da aplicação deverá fornecer uma classe que implemente a interface IScheduler, por exemplo app.Scheduler. Essa classe deverá retornar um *container*, onde será criado um agente processador, toda vez que o método getNewContainer() for chamado. Sempre que um agente processador informar ao agente escalonador que terminou sua execução, o método freeContainer() será invocado notificando a liberação do *container* que estava em utilização.

O ambiente de execução da aplicação inicia com a criação de um agente escalonador (SchedulerAgent), que é parametrizada com o número de agentes processadores que devem ser criados. O primeiro comportamento do agente escalonador (InitBehaviour) é responsável por criar os agentes processadores (ProcessorAgent) em diferentes máquinas. Esse comportamento também inicia no agente escalonador a execução do comportamento (GlobalSchedulerBehaviour) responsável por atender requisições dos agentes processadores. O comportamento GlobalSchedulerBehaviour trata as requisições por novas máquinas para a migração de execução e recebe as notificações de fim de execução dos agentes processadores. O agente escalonador conhece o número de processadores ativos, e quando não há mais processadores ativos ele encerra sua execução (FinalizeBehaviour). Considerando vários sistemas distintos, cada um deve ter seu próprio agente escalonador, embora possam compartilhar as máquinas para execução das aplicações pelo compartilhamento de *containeres*.

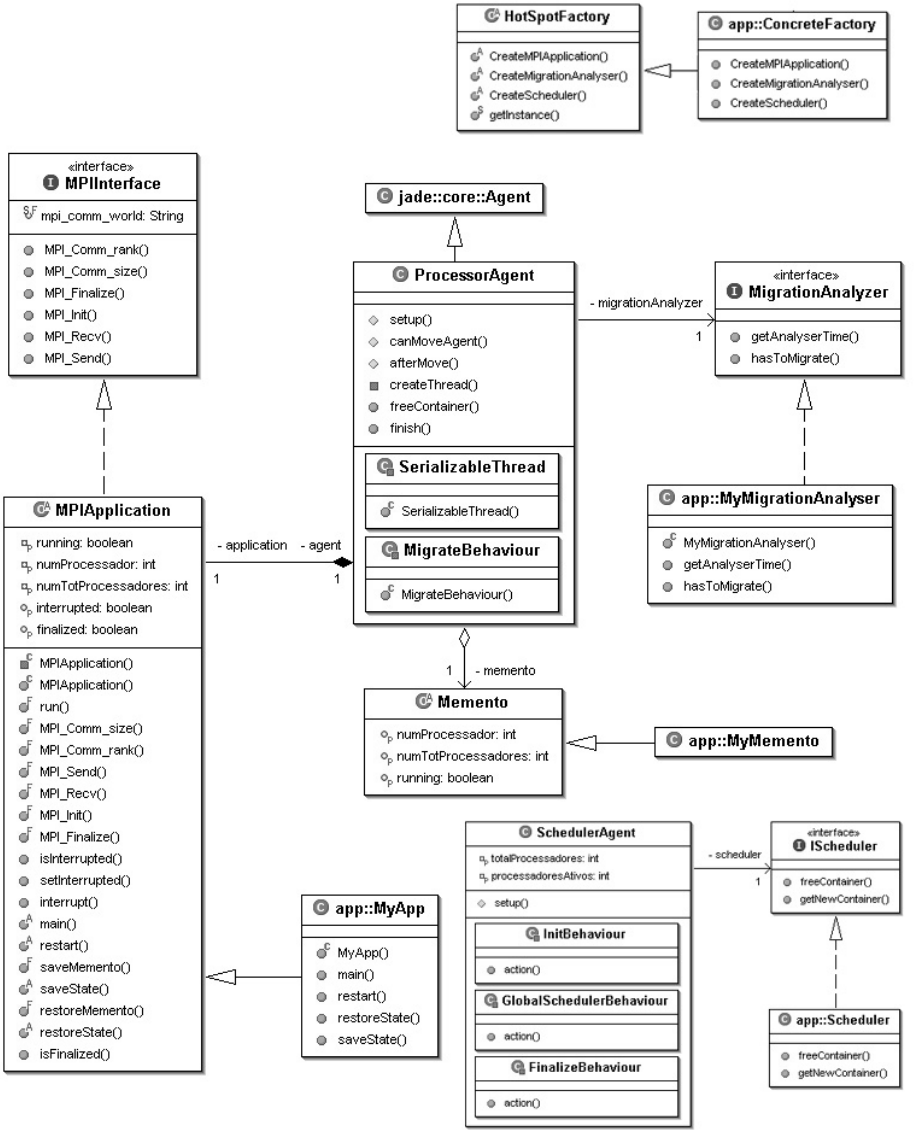

Figura 4 Diagrama de classes

5 Estudos de caso: computação de números primos

Na matemática, um número primo (ou um primo) é um número natural que possui como divisores exatamente dois números naturais distintos, onde um é o número 1 e outro o próprio número primo. Os primeiro seis primos são 2, 3, 5, 7, 11, 13 e, como demonstrado por Euclides por volta de 300 aC, existem infinitos números primos. Em 2002, Agrawal et al. [26] desenvolveram um algoritmo determinístico para verificar se um número é primo que executa em tempo polinomial. O problema de distinguir números primos de números compostos bem como o de decompor os compostos em seus fatores primos são temas importantes de estudo da aritmética. Também há um grande interesse pela utilização nas técnicas mais avançadas de criptografia [25].

Este estudo de caso trata do problema de calcular todos os primos $\leq n$, onde n é um número inteiro. O Crivo de Erastóstenes é um algoritmo clássico [27] para resolver esse problema, além de ser simples e rápido de implementar. Seu funcionamento consiste em escrever todos os inteiros de dois até n e então tentar dividir cada número subseqüente pelos primos já encontrados. O procedimento começa a partir do número dois, onde se riscam todos os números maiores do que dois que são por ele divisíveis. Em um próximo passo, seleciona-se o menor número maior do que o anterior que ainda não riscado, no caso do segundo passo esse número é o três. Assim, riscam-se todos os números maiores do que três que são por ele divisíveis. Continua-se executando esses passos até um limite, ou seja, até que todos os números divisíveis por $\sqrt{n}$ tenham sido riscados.

A complexidade deste algoritmo, apresentado na Figura 5, é determinada pelos laços nas linhas 1, 3 e 6. O primeiro laço (linha 1) percorre todos os números de 2 a n e possui a complexidade da ordem de O(n). O laço da linha 3 percorre todos os número de 2 até $\sqrt{n}$, o que dá um fator de O(n). Por fim, o laço da linha 6 é executado para cada fator primo, que pela conjetura de Gauss e Legendre totaliza aproximadamente O(log x) primos. Visto que o valor limite para x é $\sqrt{n}$, temos que a complexidade é O($\log \sqrt{n}$). Utilizando-se a aproximação de $\sqrt{n}$ = O(log n), ficamos com a complexidade de O(log log n) para o laço da linha 6. Portanto, a complexidade do algoritmo é O(n log log n), onde n é o número passado como parâmetro para o algoritmo, que é uma complexidade exponencial sobre o tamanho da entrada na representação binária. Não é um algoritmo viável para encontrar grandes números primos, mas é possível utilizá-lo para demonstrar uma técnica comum utilizada em algoritmos paralelos: resolver um caso menor do problema para acelerar a resolução do problema todo.

Esse algoritmo é facilmente paralelizável através da abordagem mestre-escravo. Reparte-se o grupo de números a serem analisados entre os diversos computadores, enquanto um computador mestre gera os números primos e os escravos analisam os grupos de números respectivos que devem ser cortados.

```
procedure CrivoErastostenes(limit);
1   for n = 2,...,limit do
2       isPrime[n] = true // assume todos os números como primos
3   for n = 2,...,sqrt(limit) do
4       if isPrime[n] then
5       // elimina múltiplos de cada primo, começando de seu quadrado
6           for i = n^2,...,limit do
7               isPrime[i] = false
8   for n = 2,...,limit do
9       if isPrime[n] then
10          print n
end CrivoErastostenes;
```

Figura 5 Pseudo-código do Crivo de Erastóstenes

Para exercitar o *framework*, uma implementação clássica em linguagem C [27] foi traduzida para ser executada na arquitetura proposta. A tradução do código de C para o *framework* foi realizada sem maiores problemas pelo fato de se utilizar o padrão MPI, mas houve necessidade de adaptar o método para suportar a migração. Essa adaptação foi realizada através de um procedimento de salvar e recuperar os dados da aplicação, e a separação do código para uma execução em duas etapas, uma para iniciar o método e outra para reiniciar o cálculo a partir de um ponto já calculado. A migração dos agentes foi testada através de requisições aleatórias de tempo, somente para verificar a funcionalidade dos agentes após uma migração. O sistema funcionou perfeitamente, porém não foram realizados testes de desempenho, pois esse não é um objetivo do trabalho.

6 Conclusões

Esta proposta de *framework* é uma alternativa para a implementação de aplicações paralelas em ambientes de sistemas multi-agentes. A utilização de Java permite a compatibilização de *hardware* heterogêneo e a plataforma JADE uma integração do sistema distribuído como uma organização. Ainda aproveitam-se as funcionalidades providas em JADE para a configuração da comunicabilidade entre as entidades da organização, diretivas de segurança e mobilidade de agentes.

O paradigma multi-agentes traz muitas abstrações, que tornam complexo o desenvolvimento de aplicações mais simples. A arquitetura proposta realiza uma abstração do mundo de agentes para facilidade de implementação focada em aplicações paralelas. A utilização de todo esse ambiente consome recursos do sistema e pode afetar o desempenho da aplicação. É necessário avaliar os benefícios de implementação da solução em comparação com a utilização de recursos do sistema. Uma proposta de trabalho é justamente melhorar a comunicação entre processos, baseando-se em outras metodologias mais

eficientes do que as atuais. O conjunto de instruções MPI implementadas nesse *framework* é suficiente para aplicações simples, mas para uma utilização real é necessário expandir a implementação da especificação. Por fim, para aproveitar melhor a visão entre os dois paradigmas, de sistemas orientados a agentes e de sistemas de aplicações paralelas, uma avaliação de integração em uma aplicação real deve revelar elementos em comum que facilite uma abstração mais refinada.

REFERÊNCIAS

[1] Markiewicz, M.E.; Lucena, C. J.. Issues on object-oriented framework Development. *ACM Crossroads*, v. 7, p. 3–9, n. 4, 2001.

[2] SETI@home, referência on-line disponível em http://setiathome.berkeley.edu/, último acesso em 15 de junho de 2006.

[3] Folding@Home, referência on-line disponível em http://www.stanford.edu/group/pandegroup/folding, último acesso em 15 de junho de 2006.

[4] Xgrid Home page, referência on-line disponível em http://www.apple.com/acg/xgrid, último acesso em 15 de junho de 2006.

[5] Message Passing Interface Forum, referência on-line disponível em http://www.mpi-forum.org, último acesso em 15 de junho de 2006.

[6] Suri, N.; Bradshaw, J. M.; Breedy, M. R.; Groth, P. T.; Hill, G. A.; Jeffers, R.. Strong mobility and fine-grained resource control in NOMADS. *Lecture Notes in Computer Science*, v. 1882, p. 2–15, 2000.

[7] Bouchenak, S.; Hagimont, D.; Krakowiak, S.; De Palma, N.; Boyer, F.. Experiences implementing efficient Java thread serialization, mobility and persistence. *Software - Practice and Experience*, v. 34, p. 355–393, n. 4, 2004 .

[8] Luck, M.; McBurney, P.; Preist, C.. A manifesto for agent technology: Towards next generation computing. *Journal of Autonomous Agents and Multi-Agent Systems*, v.9, p. 203–252, n. 3, 2004.

[9] JADE Project, referência on-line disponível em http://jade.tilab.com, último acesso em 15 de junho de 2006.

[10] See, C. S.; Ghinita, G.. Global grid connectivity using globus toolkit with Solaris™ operating system. *BluePrints*, Sun Microsystems, 2004.

[11] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley, 1995.

[12] Fünfrocken, S.. Transparent migration of Java-based mobile agents: capturing and reestablishing the state of Java programs. *Lecture Notes in Computer Science*, v. 1477, p. 26–37, 1998

[13] Sekiguchi, T.; Masuhara, H.; Yonezawas, A.. A simple extension of Java language for controllable transparent migration and its portable implementation. *Lecture Notes in Computer Science*, v. 1594, p. 211–226, 1999.

[14] Truyen, E.; Robbin, B.; Vanhaute, B.; Coninx, T.; Joosen, W.; Verbaeten, P.. Portable support for transparent thread migration in Java. *International Symposium on Agent Systems and Applications/Mobile Agents*, v.4, p. 29–43, 2000.

[15] Sakamoto, T.; Sekiguchi, T.; Yonezawa, A.. Bytecode transformation for portable thread migration in Java. *Lecture Notes in Computer Science*, v. 1882, p. 13–15, 2000.

[16] Acharya, A.; Ranganathan, M.; Salz, J.. Sumatra: A language for resource-aware mobile programs. *Lecture Notes in Computer Science*, v. 1219, p. 111–130, 1997.

[17] Suezawa, T.. Persistent execution state of a Java virtual machine. *Proceedings of the ACM 2000 conference on Java Grande*, v. 1, p. 160–167, 2000.

[18] Bouchenak, S.; Hagimont, D.. Pickling threads state in the Java system. *Proceedings of the Technology of Object-Oriented Languages and Systems Europe*, v. 33, p. 22, 2000.

[19] Illmann, T.; Krueger, T.; Kargl, F.; Weber, M.. Transparent migration of mobile agentes using the Java platform debugger architecture. *Lecture Notes in Computer Science*, v. 2240, p. 198–212, 2001.

[20] Hohl, F.. The Mobile Agent List, referência on-line disponível em http://reinsburgstrasse.dyndns.org/mal/mal.html, último acesso em 15 de junho de 2006.

[21] Baude, F.; Caromel, D.; Huet, F.; Mestre, L.; Vayssiµere, J.. Interactive and descriptor-based deployment of object-oriented grid applications. *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing HPDC-11*, v. 1, p. 93–102, 2002.

[22] Huet, F.; Caromel, D.; Bal, H. E.. A high performance Java middleware with a real application. *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, v. 1, p. 2, 2004.

[23] Baker, M.; Carpenter, B.; Ko, S.; Li, X.. MPIJava: A Java interface to MPI. *First UKWorkshop on Java for High Performance Network Computing*, Europar, 1998.

[24] Nieuwpoort, R. V. van; Maassen, J.; Wrzesinska, G.; Hofman, R.; Jacobs, C.; Kielmann, T.; Bal, H. E.. Ibis: an flexible and efficient java-based grid programming environment. *Concurrency and Computation: Practice and Experience*, v. 17, p. 1079–1107, 2005.

[25] Diffie, W.; Hellmann, M. E.. New directions in Cryptography. *IEEE Transactions on Information Theory*, v. IT-22, p. 644-654, IEEE Computer Society, 1976.

[26] Agrawal, M.; Kayal, N.; Saxena, N.. PRIMES is in P. *Annals of Mathematics*, v. 160, p. 781–793, 2004.

[27] Quinn, M. J.. *Parallel Computing: Theory and Practice*, McGraw Hill, 1994.